Definição e Implementação da linguagem *Coffee*

Departamento de Computação
Universidade Federal de Ouro Preto
Prof. José Romildo Malaquias
31 de agosto de 2020

Resumo

Coffee é uma pequena linguagem de programação usada para fins didáticos na aprendizagem de técnicas de construção de compiladores.

A documentação e implementação de *Coffee* será realizada de forma colaborativa pelos participantes do curso.

Sumário

1 A linguagem *Coffee*

Andrew Appel apresenta em seu livro *Modern Compiler Implementation in ML* uma pequena linguagem de programação para fins didáticos chamada *Tiger*. Nesta disciplina vamos considerar uma linguagem semelhante, que chamaremos de *Coffee*, baseada em *Tiger*, porém com algumas diferenças sintáticas e semânticas.

Coffee é uma linguagem de programação bastante simples que será usada para praticarmos a implementação de um compilador com aplicação das técnicas discutidas nas aulas.

As construções da linguagem serão detalhadas no decorrer o curso. Começaremos com uma versão básica, e oportunamente serão apresentadas versões mais aprimoradas, com novas construções.

Coffee é uma linguagem imperativa com tipagem estática. Seus tipos de dados básicos são:

- `bool`: valores lógicos
- `int`: valores inteiros de precisão fixa
- `real`: valores reais (números em ponto flutuante de dupla precisão)
- `string`: cadeias de caracteres

2 O projeto

2.1 Estrutura do projeto

O projeto será desenvolvido na linguagem OCaml usando a ferramenta dune para automatização da compilação. O projeto usa algumas ferrametnas e bibliotecas externas:

ocamllex Ocamllex é um gerador de analisador léxico. Ele produz um analisadores léxicos (em OCaml) a partir de conjuntos de expressões regulares com ações semânticas associadas. É distribuído junto com o compilador de OCaml.

menhir Menhir é um gerador de analisador sintático. Ele transforma especificações gramaticais de alto nível, decoradas com ações semânticas expressas na linguagem de programação OCaml, em analisadores sintáticos, também expressos em OCaml.

dune Dune é um sistema de compilação para OCaml (e Reason).

ppx_import É uma extensão de sintaxe que permite extrair tipos ou assinaturas de outros arquivos de interface compilados

ppx_deriving É uma extensão de sintaxe que facilita geração de código baseada em tipos em OCaml

ppx_expect É uma extensão de sintaxe para escrita de testes em OCaml

camomile Camomile é uma biblioteca unicode para OCaml.

O código é organizado segundo a estrutura de diretórios mostrada na figura 1.

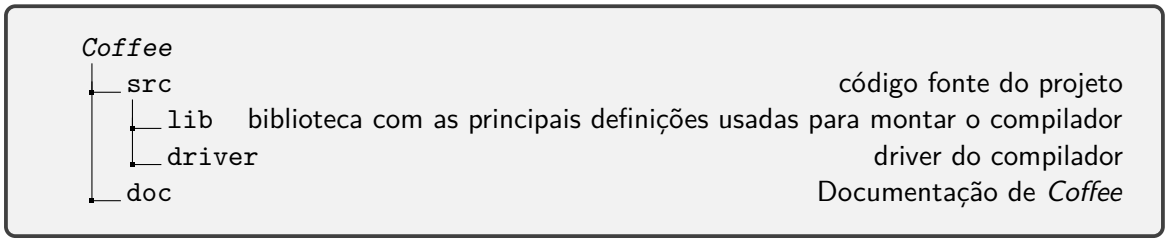

Figura 1: Estrutura de diretórios do projeto do compilador

Existem outros diretórios gerados automaticamente que não são relevantes nesta discussão e por isto não foram mencionados. Caso algum ambiente de desenvolvimento integrado (IDE) seja usado, provavelmente haverá alguns arquivos e diretórios específicos do ambiente e que também não são mencionados.

Os arquivos `src/lib/dune` e `src/driver/dune` contêm as especificações do projeto esperada pelo dune. Neles são indicadas informações como nome do projeto, dependências externas e flags necessários para a compilação da biblioteca e da aplicação.

2.2 Módulos importantes

Alguns módulos importantes no projeto são mencionados a seguir. Alguns já estão prontos, e outros deverão ser criados ou modificados pelo participante nas atividades do curso.

- O módulo `Absyn` contém os tipos que representam as árvores sintáticas abstratas para as construções da linguagem fonte.
- O módulo `Lexer` contém as declarações relacionadas com o analisador léxico do compilador.

  O analisador léxico (módulo `Lexer`) é gerado automaticamente pelo `ocamllex`. A especificação léxica é feita no arquivo `src/lib/lexer.mll` usando expressões regulares.
- O módulo `Parser` contém as declarações relacioandas com o analisador sintático do compilador.

  O analisador sintático (módulo `Parser`) é gerado automaticamente pelo `menhir`. A especificação sintática é feita no arquivo `src/lib/parser.mly` usando uma gramática livre de contexto.
- O módulo `Semantic` contém declarações usadas na análise semântica e geração de código do compilador.
- O módulo `Symbol` contém declarações que implementam um tipo usado para representar nomes de identificadores, e discutido posteriormente.
- O módulo `Environment` contém declarações para a manipulação dos ambientes (ás vezes também chamados de contexto) de compilação. Estes ambientes são representados usando tabelas de símbolos.
- O módulo `Types` contém declarações para a representação interna dos tipos da linguagem fonte.
- O módulo `Error` contém declarações usadas para reportar errors detectados pelo compilador durante a compila

- O módulo `Location` contém declarações usadas para representação de localizações de erros no código fonte, importantes quando os erros forem reportados.
- O módulo `Driver` é formado por declarações, incluindo a função `main`, ponto de entrada para execução do compilador.

2.3 Acrescentando uma nova construçao de *Coffee*

Ao acrescentar uma nova construção na implementação da linguagem, procure seguir os seguintes passos:

- Se necessário defina um novo construtor de dados no tipo `Types.t` para representar algum tipo da linguagem *Coffee* que ainda não faça parte do projeto.
- Se necessário acrecente ao ambiente inicial (no módulo `Environment.env`) a representação de quaisquer novos tipos, variáveis ou funções que façam parte da biblioteca padrão de *Coffee*.
- Defina os novos construtores de dados necessárias para representar a árvore abstrata para a construção no tipo `Absyn.t`.
  - definir os campos necessários para as sub-árvores da árvore abstrata,
  - extender a função `to_tree` que permite converter para uma árvore de strings, útil para visualização gráfica da árvore abstrata.
- Extenda a função de análise semântica `Semantic.semantic` para tratar a nova construção.
- Declare quaisquer novos símbolos terminais e não-terminais na gramática livre de contexto da linguagem que se fizerem necessários para as especificações léxica e sintática da construção.
- Acrescente as regras de produção para a construção na gramática livre de contexto da linguagem, tomando o cuidado de escrever ações semânticas adequadas para a construção da árvore abstrata correspondente. Se necessário use declarações de precedência de operadores.
- Se necessário acrescente regras léxicas que permitam reconhecer os novos símbolos terminais na especificação léxica da linguagem.

3 Mensagens de erro

O projeto contém algumas funções para reportar erros encontrados durante a compilação. Estas funções fazem parte do módulo `Error` e serão comentadas a seguir.

Em todo compilador é desejável que os erros encontrados sejam reportados com uma indicação da localização do erro, acompanhada por uma mensagem explicativa do problema ocorrido. Para tanto torna-se necessário manter a informação da localização em que cada frase do programa (cada nó da árvore abstrata construída para representar o programa) foi encontrada. O módulo `Location` contém algumas definições relacionadas com estas localizações.

Neste projeto as localizações no código fonte são representadas pelo tipo `Location.t`, que leva em consideração as posições no código fonte onde a frase começou e terminou.

Cada uma destas posições é do tipo `Lexing.position`. O módulo `Lexing` faz parte da biblioteca padrão do OCaml e será extensivamente usado nas implementações dos analisadores léxico e sintático. O tipo `Lexing.position` contém as seguintes informações:

- a indicação da unidade (arquivo fonte) sendo compilada,
- o número da linha, e
- o número da coluna

A função `Error.error` e outras similares encontradas no módulo `Error` devem ser usadas para emissão de mensagens de erro. Esta função recebe como argumentos a localização do erro, a mensagem de formatação de diagnóstico, e possivelmente argumentos complementares de acordo com a mensagem.

4 Aspectos léxicos

4.1 Brancos e comentários

Ocorrências de **caracteres brancos** (espaço, tabulação horizontal, nova linha) e comentários entre os símbolos léxicos são ignorados.

Comentários são ignorados pelo compilador e podem ser úteis como anotações sobre o programa para alguém que estiver lendo ou modificando o programa. Os comentários podem ser:

- **comentários de linha**, que em *Coffee* começam com o caractere `#` e se estendem até o final da linha.
- **comentários de bloco**, que são delimitados pelas sequências de caracteres `{#` e `#}` e podem ser **aninhados**.

Exemplos de comentários:

```
# isto é um comentário de linha
```

```
{# isto é um
   comentário de bloco #}
```

```
{# isto é um
   comentário de bloco {# aninhado #}
   percebeu? #}
```

Os seguintes comentários estão incorretos, pois não foram terminados:

```
{# Este comentário de bloco
não terminou!
```

```
{# Este comentário de bloco {# com aninhamento #}
   também não terminou!
```

4.2 Literais

4.2.1 Literais inteiros

Os **literais inteiros** são formados por uma sequência de um ou mais dígitos decimais.

São exemplos de literais inteiros:

```
2014
872834
0
0932
```

Não há nenhum literal inteiro negativo.

4.2.2 Literais reais

Os **literais reais** correspondem a números em ponto flutuante possivelmente em notação científica. São formados por uma parte inteira, seguida de uma parte decimal e/ou uma parte exponencial.

A parte inteira é formada por uma sequência de um ou mais dígitos decimais.

A parte decimal é formada pelo caracter `.`, seguido de uma sequência de um ou mais dígitos decimais.

A parte exponencial representa uma potência de dez é formada por um dos caracteres `e` ou `E`, seguido opcionalmente dos caracteres `+` ou `-`, seguido de uma sequência de um ou mais dígitos decimais.

São exemplos de literais reais:

```
20.14
0.0872834
123.456e12
5632.003E-15
77e100
```

Não há nenhum literal real negativo.

4.2.3 Literais lógicos

Os **literais lógicos** são `true` (verdadeiro) e `false` (falso).

4.2.4 Literais string

Os **literais string** são formados por uma sequência de caracteres delimitada por aspas (`"`). Na sequência de caracteres o caracter `\` é especial e inicia uma sequência de escape. Uma sequência de escape representa um caracter de acordo com a tabela a seguir.

sequência de escape	**descrição**
`\\`	`\`
`\"`	`"`
`\t`	tabuação horizontal
`\n`	nova linha
`\r`	retorno de carro
`\b`	retrocesso
`\`*`ddd`*	caracter de código *ddd*, sendo *ddd* uma sequências de 3 dígitos decimais

Estas são as únicas sequências de escape válidas.

São exemplos de literais string:

```
"Tiger"
"Bom dia, Brasil!"
"B"
"\065 = \"B\""
""
"abc\tDEF\nGHI\\JKL\"mno\065ok"
```

Não são exemplos de literais string:

```
# invalid escape sequence in string literal
"abc\kdef"
"\64 is not ok"

# unclosed string literal
"abc
```

4.2.5 Identificadores

Identificadores são sequências de letras maiúsculas ou minúsculas, dígitos decimais e sublinhados (_), começando com uma letra. Letras maiúsculas e minúsculas são distintas em um identificador.

Identificadores são usados para nomear entidades usadas em um programa, como tipos, funções e variáveis.

São exemplos de identificadores:

```
peso
idadeAluno
alfa34
primeiro_nome
```

Não são exemplos de identificadores:

```
__peso
idade do aluno
34rua
primeiro-nome
```

4.2.6 Operadores, sinais de pontuação e palavras-chave

Os seguintes operadores podem ser usados em *Coffee*:

```
+ - * / % ^
= <> > >= < <=
&& ||
:=
```

São sinais de pontuação em *Coffee*:

```
( ) , ; :
```

As palavras-chave de *Coffee* são:

```
var
if then else
while do break
let in end
```

Palavras-chave são reservadas, isto é, não podem ser usadas como identificadores. Todas as palavras-chave são escritas com letras minúsculas.

5 Símbolos

Linguagens de programação usam **identificadores** para nomear entidades da linguagem, como tipos, variáveis, funções, classes, módulos, etc.

Símbolos léxicos (também chamados de símbolos terminais ou *tokens*) que são classificados como identificadores tem um valor semântico (atributo) que é o nome do identificador. A princípio o valor semântico do identificador pode ser representado por uma cadeia de caracteres (tipo `string` do OCaml). Porém o tipo `string` tem algumas incoveniências para o compilador:

- Geralmente o mesmo identificador ocorre várias vezes em um programa. Se cada ocorrência for representada por uma string (ou seja, por uma sequência de caracteres), o uso de memória poderá ser grande.
- Normalmente existem dois tipos de ocorrência de identificadores em um programa:
  - uma declaração do identificador, e
  - um ou mais usos do identificador já declarado.

  Durante a compilação cada ocorrência de uso de um identificador deve ser associada com uma ocorrência de declaração. Para tanto os identificadores devem ser comparados para determinar se são iguais (isto é, se tem o mesmo nome). O uso de strings é ineficiente, pois pode ser necessário comparar todos os caracteres da string para determinar se elas são iguais ou não.

Por estas razões o compilador utiliza o tipo `Symbol.symbol` para representar os nomes dos identificadores. Basicamente mantém-se uma tabela *hash* onde os identificadores são colocados à medida que eles são encontrados. Sempre que o analisador léxico encontrar um identificador, deve-se verificar se o seu nome já está na tabela. Em caso afirmativo, usa-se o símbolo correspondente que já se encontra na tabela. Caso contrário cria-se um novo símbolo, que é adicionado à tabela associado ao nome encontrado, e é usado pelo analisador léxico como valor semântico do *token*.

A implementação de `Symbol.symbol` associa-se a cada novo símbolo um número inteiro diferente. A comparação de igualdade de símbolos se resume a uma comparação (muito eficiente) de inteiros, já que o mesmo identificador estará sempre sendo representado pelo mesmo símbolo (associado portanto ao mesmo número inteiro).

A função `Symbol.symbol` cria um símbolo a partir de uma string.

6 O analisador léxico

O módulo `Lexer` contém as declarações que implementam o analisador léxico do compilador. Este módulo será gerado automaticamente pela ferramenta `ocamllex`.

A especificação da estrutura léxica da linguagem fonte é feita no arquivo `src/lib/lexer.mll` usando expressões regulares. Consulte a documentação do `ocamllex` para entender como fazer a especificação léxica.

Os analisadores léxico e sintático vão se comunicar durante a compilação, pois os tokens obtidos pelo analisador léxico serão consumidos pelo analisador sintático. Ou seja, os tokens são os símbolos terminais da gramática usada pelo gerador de analisador sintático. Para manter a consistência dos analisadores léxico e sintático os símbolos terminais (tokens) são declarados na gramática livre de contexto do `menhir`, no arquivo `src/lib/parser.mly`. Consulte a documentação do `menhir` para entender como escrever a gramática livre de contexto.

7 Sintaxe

A sintaxe de todas as construções de *Coffee* é apresentada na gramática livre de contexto que se segue.

Program → *Exp*	programa
Exp → `litint`	literais
Exp → `litreal`	
Exp → `litbool`	
Exp → `litstring`	
Exp → *Var*	variável
Exp → `-` *Exp*	operações aritméticas
Exp → *Exp* `+` *Exp*	
Exp → *Exp* `-` *Exp*	
Exp → *Exp* `*` *Exp*	
Exp → *Exp* `/` *Exp*	
Exp → *Exp* `%` *Exp*	
Exp → *Exp* `^` *Exp*	
Exp → *Exp* `=` *Exp*	operações relacionais
Exp → *Exp* `<>` *Exp*	
Exp → *Exp* `>` *Exp*	
Exp → *Exp* `>=` *Exp*	
Exp → *Exp* `<` *Exp*	
Exp → *Exp* `<=` *Exp*	
Exp → *Exp* `&&` *Exp*	operações lógicas
Exp → *Exp* `||` *Exp*	
Exp → *Var* `:=` *Exp*	atribuição
Exp → `id` `(` *Exps* `)`	chamada de função
Exp → `if` *Exp* `then` *Exp* `else` *Exp*	expressões condicionais
Exp → `if` *Exp* `then` *Exp*	
Exp → `while` *Exp* `do` *Exp*	expressão de repetição
Exp → `break`	
Exp → `let` *Decs* `in` *Exp*	expressão de declaração
Exp → `(` *ExpSeq* `)`	expressão sequência
Exps →	sequência de expressões separadas por `,`
Exps → *Exp* *ExpsRest*	

$$
\begin{array}{rcll}
\mathit{ExpsRest} & \rightarrow & & \\
\mathit{ExpsRest} & \rightarrow & \texttt{,}\ \ \mathit{Exp}\ \ \mathit{ExpsRest} & \\
 & & & \\
\mathit{ExpSeq} & \rightarrow & & \text{sequência de expressões separadas por } \texttt{;} \\
\mathit{ExpSeq} & \rightarrow & \mathit{Exp}\ \ \mathit{ExpSeqRest} & \\
\mathit{ExpSeqRest} & \rightarrow & & \\
\mathit{ExpSeqRest} & \rightarrow & \texttt{;}\ \ \mathit{Exp}\ \ \mathit{ExpSeqRest} & \\
 & & & \\
\mathit{Var} & \rightarrow & \texttt{id} & \text{variável simples} \\
 & & & \\
\mathit{Decs} & \rightarrow & \mathit{Dec} & \text{sequência de declarações} \\
\mathit{Decs} & \rightarrow & \mathit{Dec}\ \ \mathit{Decs} & \\
 & & & \\
\mathit{Dec} & \rightarrow & \mathit{DecVar} & \text{declaração} \\
 & & & \\
\mathit{DecVar} & \rightarrow & \texttt{var}\ \ \texttt{id}\ \ \texttt{:}\ \ \texttt{id}\ \ \texttt{=}\ \ \mathit{Exp} & \text{declaração de variável} \\
\mathit{DecVar} & \rightarrow & \texttt{var}\ \ \texttt{id}\ \ \texttt{=}\ \ \mathit{Exp} &
\end{array}
$$

Observe que um **programa** em *Coffee* é uma expressão.

A precedência relativa e a associatividade dos operadores é indicada pela tabela a seguir, em ordem decrescente de precedência.

operadores	**associatividade**
`-` (unário)	
`^`	direita
`*`, `/`, `%`	esquerda
`+`, `-` (binário)	esquerda
`=`, `<>`, `>`, `>=`, `<`, `<=`	
`&&`	esquerda
`||`	esquerda
`:=`	
`then`, `else`, `do`, `in`	direita